gt4python

Working with Python inside Glamorous Toolkit

Veit Heller (veit.heller@feenk.com), svenvc (in spirit)

Agenda

1. Working with *Python* in GT
2. Inspecting and playing
3. Three case studies

Python Examples I

```
x = 2 ** 30
```

Python @ localhost:7056

```
import datetime

datetime.datetime.fromtimestamp(x)
```

Python @ localhost:7056

```
print("Where is this going?")
```

Python @ localhost:7056

<an unnamed volatile knowledgebase>

Python Examples II

Python @ localhost:7056

```
import random

from collections import Counter

counter = Counter()

for i in range(100):
    counter[random.randint(0, 10)] += 1

counter
```

Python @ localhost:7056

<an unnamed volatile knowledgebase>

Python Examples III

```
filePath := FileLocator▸ imageDirectory▸ pathString
```

```
print(filePath)
```

Python @ localhost:7056

<an unnamed volatile knowledgebase>

Python Debugger

```
pbApplication := PBApplication new.
pbApplication settings serverDebugMode: true.
pbApplication start.
PBApplication uniqueInstance: pbApplication
```

Python @ localhost:7056

```
def i_recurse(n):
    if n % 10 == 0:
        pbbreak()
    i_recurse(n+1)

i_recurse(1)
```

```
pbApplication := PBApplication new.
pbApplication settings serverDebugMode: true.
pbApplication start.
PBApplication uniqueInstance: pbApplication
```

<an unnamed volatile knowledgebase>

PythonBridge custom views for pandas DataFrame and Series

One of the most well know Python ▸ libraries is pandas ▸ , a data analysis and manipulation tool.

The following Python source code file adds a number of remote Phlow views for the main DataFrame and Series classes.

This page explains the fast way to get started, as a user of this feature. To learn more about the internals, consult Adding gtViews to the existing Python pandas DataFrame and Series class ▸ .

The file location is defined next, inspect it to see the contents.

```
pandasViews := FileLocator▸ gtResource▸ / 'feenkcom' / 'gt4python' / 'data' /
'python' / 'view_pandas.py'
```

Given the path above, we can load the code programmatically into the global `PBApplication` ▸ instance (which should already be running with pandas installed).

When necessary:

```
PBApplication isRunning ifFalse: [ PBApplication start ].
PBApplication uniqueInstance installModule: 'pandas'.
```

```
PBApplication do: [ :application |
    application newCommandStringFactory
        script: pandasViews contents;
        sendAndWait ]
```

You can test the views by loading and inspecting a DataFrame from an example CSV file.

```
moviesCsvPath := FileLocator gtResource / 'feenkcom' / 'gtoolkit-demos' /
'data' / 'imdb' / 'Movies.csv'
```

```
moviesCsvPathString := moviesCsvPath resolve pathString
```

Python @ localhost:7056

```
import pandas
pandas.read_csv(moviesCsvPathString)
```

▸ 2 explicit references

Glamorous Toolkit Book

Inspecting Python objects with custom inspectors

Glamorous Toolkit works with other runtimes. For example, we can work with Python ▸ through PythonBridge ▸ . But what might be less obvious is that we can also extend the inspector using Python.

To exemplify how this works, let's consider exploring a movie collection defined in this CSV:

```
csvFile := (FileLocator▸ gtResource▸ / 'feenkcom' / 'gtoolkit-demos' / 'data' /
'imdb' / 'Movies.csv') fullName.
```

We load it with pandas, and to do that we first set up the Python runtime by installing the pandas module. (NB: You might need to install pipenv first.)

```
PBApplication▸ isRunning▸ ifFalse: [ PBApplication▸ start▸ ].
PBApplication▸ uniqueInstance▸ installModule: 'pandas'.
```

Ok, now that's done. Next, we define views as extensions to the movie collection entities:

Python @ localhost:7056

```
import pandas
from gtoolkit_bridge import gtView

class Movie:
    def __init__(self, series):
        self.series  = series

    @gtView
    def gtViewDescription(self, builder):
        text = builder.textEditor()
        text.title("Description")
        text.priority(30)
        text.setString(str(self.series))
        return text

    @gtView
    def gtViewDetails(self, builder):
        clist = builder.columnedList()
        clist.title("Details")
        clist.priority(20)
        clist.items(lambda: list(self.series.index))
        clist.column('Key', lambda each: each)
        clist.column('Value', lambda each: str(self.series[each]))
        clist.set_accessor(lambda each: self.series[each])
        return clist

class MovieCollection:
    def __init__(self, dataFrame):
        self.df = dataFrame

    def size(self):
        return len(self.df.index)

    def movieAtPosition(self, index):
        return Movie(self.df.loc[index])

    def directors(self):
        values = self.df["Directors"].astype(str).unique()
        values.sort()
```

Visualizing tokenization

OpenAI publishes tiktoken, an open-source Python library that tokenizes input strings as it is done internally for their product. This is useful for estimating costs as well as ensuring inputs stay within certain limits (context windows etc.). `gt4openai` also relies on this library internally, for instance to estimate the cost of a fine-tuning job.

To evaluate the result of tokenizing with various models, you can work directly with the tiktoken library. But you can also work with a small wrapper we created that helps us visualize the results. To this end, first install the `gtoolkit_tiktokenize` module:

```
PBApplication▸ isRunning▸ ifFalse: [ PBApplication▸ start▸ ].
PBApplication▸ uniqueInstance▸ installModule: 'gtoolkit_tiktokenize'
```

And then just execute the Python script:

Python @ localhost:7056

```
import gtoolkit_tiktokenize

model = "gpt-4"
string = "<|im_start|>system<|im_sep|>You are a helpful assistant<|im_end|><|
im_start|>assistant<|im_sep|>How can I help you?<|im_end|><|im_start|>user<|
im_sep|>Can you add two and two for me?<|im_end|>"
gtoolkit_tiktokenize.tokenize(string, model)
```

The equivalent in Pharo code is as follows:

```
GtLlmTokenizer new
    tokenizeMessages: {GtLlmSystemMessage new content: 'You are a helpful
assistant'.
            GtLlmAssistantMessage new content: 'How can I help you?'.
            GtLlmUserMessage new content: 'Can you add two and two for me?'}
    usingModel: 'gpt-4'
```

▸ 1 explicit reference

gt4llm-lepiter

Exploring the execution of a Python LampSort algorithm

This tutorial shows how to analyse Python code using GT's PythonBridge. In particular, we'll look at a simple sorting algorithm and mold our tools to explain how the algorithm works.

Setup

First make sure GT's PythonBridge is up and running. Let's define a small list of random numbers.

Python @ localhost:7056

```
import random
numbers = random.sample(range(100), 10)
```

The indicator in the Python snippet above should show an active Python connection.

We can now copy our tutorial files into the `PythonBridgeRuntime` directory next to our GT image.

```
files := FileLocator▸ gtResource▸ / 'feenkcom' / 'gt4python' / 'data' /
'python' childrenMatching: 'lampsort*.py'.
files do: [ :each | each copyTo: PBPlatform▸ current▸ workingDirectory / each
basename ].
PBPlatform▸ current▸ workingDirectory
```

#LampSort

Now we can open a view on the Python files inside into the `PythonBridgeRuntime` directory next to our GT image which now contains our tutorial files. Let's look at `lampsort0.py` which is our orginal version.

```
GtLSPPythonModel onDirectory: PBPlatform current workingDirectory
```

The above tool uses [LSP](https://en.wikipedia.org/wiki/Language_Server_Protocol). For optimal usability during editing you should install [pyright](https://github.com/microsoft/pyright).

`LampSort` is an object that wraps a data list and sorts it in place. The LampSort algorithm is a non-recursive implementation of [QuickSort](https://en.wikipedia.org/wiki/Quicksort).

It works by partitioning the list into intervals around a pivot, where the left interval contains elements smaller than the pivot and the right interval elements larger than the pivot. Intervals are partitioned further until they contain one element or are empty, in which case they are sorted by default.

The algorithm starts from a set containing one interval covering the whole list. It is done when this set is empty. Range objects are used to represent intervals.

Let's make sure our implementation works.

Python @ localhost:7056

```
import lampsort0
result = lampsort0.LampSort(numbers.copy()).sort()
assert result == sorted(numbers)
result
```

#Tracing

If you look carefully at the LampSort implementation you'll notice that there are method for each individual step, with a meaningful name and with relevant arguments (the data list of elements is never passed as an argument, it is an instance variable).

Thank you!

Questions?

https://gtoolkit.com